"Express Mail" mailing label number:

EL604108865US

DISTRIBUTED LEAST CHOICE FIRST ARBITER

Nils Gura
Hans Eberle

RELATED APPLICATIONS

This application is a continuation in part of application number 09/540,729, filed March 31, 2000, entitled "Least Choice First Arbiter", naming as inventors Hans Eberle and Nils Gura, which application is incorporated herein by reference in its entirety.

COPYRIGHT NOTICE

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates to systems having shared resources and more particularly to arbitrating multiple requests for multiple resources is such systems.

Description of the Related Art

Systems having shared resources are common. In many such systems, arbiters have to schedule usage of the shared resources to prevent conflicts resulting from requests for simultaneous access to the same shared resource. One example of such a system having shared resources susceptible to conflicts is a crossbar switch having multiple input ports and multiple output ports in which input ports make requests for connection to the output ports. Each requester or input port sends a request for an output port or a set of requests for multiple output ports to an arbiter. A particular output port may be requested by multiple input ports at the same time. Assuming an output port can be allocated to only one input port at a time, an arbitration decision is made to award the output port to one of the requesters. The arbiter chooses the requests to be granted such that resources (the output ports) are allocated to requesters in a conflict-free way. Thus, certain requests are serviced by being granted an output port and certain requests may be left unserviced in that they are denied an output port. However, the choice of which requests to grant may lead to under-utilization of the resources since some requests may be denied.

Another example of a system having shared resources is a computer system in which multiple processors are coupled to multiple memories. Assume that each processor has access to all of the memories and each memory can only be accessed by one processor at a time. When multiple processors request access to the same memory at the same time an arbitration decision has to be made as to which processor gets to access the memory in question.

While numerous arbitration schemes have been developed to try and provide fair and efficient allocation of system resources for scheduling problems that involve multiple requesters requesting multiple shared resources such as the crossbar switch or multi-processor system described above, it would be desirable to have an improved arbitration scheme that provides for high aggregate usage of the shared resources while still providing a minimum level of fairness.

SUMMARY OF THE INVENTION

Accordingly, the invention provides in one embodiment an arbiter that prioritizes requests based on the number of requests made. The highest priority is given to the requester that has made the fewest number of requests. That is, the requester with the fewest requests (least number of choices) is chosen first. Requesters with more requests have more choices than requesters with fewer requests and, therefore have a reasonable likelihood of being granted one of their outstanding requests if considered after those requesters with fewer requests. Resources may be scheduled sequentially or in parallel. In order to prevent starvation, a round robin scheme may be used to allocate a resource to a requester, prior to issuing grants based on requester priority.

If multiple grants are received by a requester, the requester may prioritize the received grants according to resource priority, that is, according to the number of requests received by the resource. The resource receiving the fewest requests has the highest priority. Utilizing a strategy that considers fewest choices first, increases the number of granted requests and results in higher aggregate resource usage when compared with other arbitration schemes. A round robin scheme may also be utilized to prevent starvation when accepting grants. Resources may be allocated sequentially or in parallel.

In one embodiment, the invention provides a method for operating a distributed arbiter for a plurality of resources, that includes receiving requests at one of the resources from a plurality of requesters. The resource grants the one resource to one of the requesters according to requester priorities, the requester priorities being inversely related to a number of requests made respectively, by the requesters.

In another embodiment, a distributed arbiter includes a plurality of requesters and a plurality of resources coupled to the requesters through a transport mechanism, such as a bus or switch. Each requester is coupled to provide to each resource requested by the respective requester with a request indication. A requested resource is responsive to a plurality of requests to selectively grant one of the requests according to requester priorities, the requester priorities being inversely related to a number of requests being made by respective requesters.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings wherein use of the same reference symbols in different drawings indicates similar or identical items.

Fig. 1 shows a crossbar switch which can exploit one or more embodiments of the present invention.

Fig. 2 shows a multi-processor system which can exploit one or more embodiments of the present invention.

Figs. 3A-3C illustrate operation of one embodiment of the invention where priorities are determined according to a number of requests each requester makes.

Fig. 4 illustrates a block diagram of a hardware implementation of an arbiter according to one embodiment of the present invention.

Figs. 5A-5C illustrate operation of one embodiment of the invention where priorities are determined according to a number of requests each resource receives.

Fig. 6 illustrates a switch that utilizes a distributed embodiment of the present invention.

Fig. 7 illustrates the requests and grants passed by requesters and resources in an exemplary distributed system.

Fig. 8 illustrates exemplary vectors of requests sent from a requestor to a resource in an exemplary distributed system.

Fig. 9A- 9D illustrate operation of a sequential distributed arbiter, also described in Appendix C, according to an embodiment of the invention.

Fig. 10 illustrates data flow between requesters and resources in a distributed arbiter according to an embodiment of the invention.

Fig. 11A- 11B illustrate operation of a parallel distributed arbiter, also described in Appendix D, according to an embodiment of the invention.

Fig. 12 illustrates data flow between requesters and resources in a parallel distributed arbiter according to an embodiment of the invention.

Fig 13 illustrates a bus based distributed system suitable for using an embodiment of the invention.

Fig. 14 illustrates the timing of operation of an embodiment of a sequential distributed arbiter utilizing a bus structure shown in Fig. 13.

Fig. 15 illustrates the timing of operation of an embodiment of a parallel distributed arbiter utilizing a bus structure shown in Fig. 13.

Fig. 16A - 16F illustrate another example of operation of an embodiment of a parallel distributed arbiter.

Fig. 17 illustrates an open collector bus structure for resolving priority in a distributed arbiter.

Fig 18A and 18B show high level block diagrams of a centralized and distributed arbiter.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

Referring to Fig. 1, one embodiment of the invention is illustrated in which arbiter 101 schedules usage of shared resources, i.e., output ports 103, 105 and 107 among input port requesters, 109, 111 and 113. Crossbar switch 115 forwards packets from its input ports, 109, 111 and 113 to its output ports 103, 105 and 107. Each input port can hold multiple packets destined for different output ports. The switch schedule to be calculated should connect input and output ports in such a way that as many packets as possible can be forwarded simultaneously, thus trying to maximize usage of shared resources. Each requester sends a set of requests to arbiter 101, which then chooses the requests to be granted such that resources are allocated to requesters in a conflict-free way.

In one embodiment of the invention, the arbiter operates in a synchronous manner in that the arbiter receives request signals 117 for shared resources at the same time from the various nodes 121, 123 and 125. Scheduling happens synchronously in that grant signals 119 are sent at the same time and the usage interval for each resource has the same length. Scheduling may be further constrained in that only one requester can use a particular resource at the same time. When developing an arbitration scheme the main goal typically is to achieve high aggregate usage of the resources while still providing a minimum level of fairness, mainly in the sense that starvation of individual requests is prevented.

In one embodiment, the arbiter makes its choices by prioritizing requesters based on the number of their requests. The highest priority is given to the requester with the fewest number of outstanding requests, and the lowest priority is given to the requester with the highest number of outstanding requests. Resources are scheduled one after the other in that a resource is allocated to the requester with the highest priority first. That way the requester with the least number of choices is chosen first. Thus, priority is inversely related to the number of requests being made. Requesters with many requests have more choices than requesters with fewer requests and, therefore, can be considered later and still have a reasonable likelihood of being granted one of their outstanding requests. That strategy increases the number of granted requests and, with it, results in higher aggregate usage when compared with other arbitration schemes. In one preferred embodiment, the number of outstanding requests is based only on the resources that have not yet been scheduled. That is, the number of outstanding requests is recalculated whenever a resource has been scheduled.

With reference again to Fig. 1, an example illustrates operation of one aspect of an embodiment of the least choice first arbiter. In Fig. 1, it can be seen that node 121, which is coupled to input port 109, has one request 122 for output port 103. Node 123, which is coupled to input port 111, has two requests 124, one request for output port 105 and one request for output port 107, respectively. Node 125, which is coupled to input port 113, has three requests 126, one request for each of the output ports. The arbiter receives those requests from the various nodes and prioritizes the request from node 121 (input port 109) as the highest priority request since that requester has only one request. Node 123 is prioritized as the next highest priority requester since it has two requests and node 125 is the lowest priority requester since it has three requests.

Based upon priority alone, arbiter 101 grants node 121 its request for output port 103 first. Note that while the node attached to the input port is actually making requests, for ease of description, the input ports themselves may be described herein as having requests. After arbiter 101 grants node 121 its request, priorities are recalculated. Both node 123 and node 125 have the same priority after output port 103 is scheduled, since they both have two requests (requests for output port 103 are no longer considered since that output port has been scheduled). The arbiter can now grant one of the requests of node 123 and node 125. For example, if the arbiter grants node 123 its request for output port 105, the arbiter then grants node 125 its request for output port 107. Because the arbiter serviced the requester having the least number of choices first, the number of granted requests can be maximized. To illustrate that, if another arbitration scheme had first granted node 125 its request for output port 103, a total of only two requests would have been granted. Note that the arbiter has not considered fairness in allocating the resources in the example given.

Another example where such an arbitration scheme may be utilized is shown in Fig. 2, which is a multi-bus interconnection structure 200 functioning as a transport mechanism for connecting multiple processors 201, 203 and 205 (also referred to in Fig. 2 as P0, P1 and P2) with multiple memories 207, 209 and 211 (also referred to in Fig. 2 as M0, M1 and M2). Each processor can have multiple outstanding transactions such as read and write operations. Similar to the switch embodiment described in relation to Fig. 1, the bus schedule to be calculated connects processors and memories in a conflict-free way, ideally, such that at a given time as many transactions as possible can be executed in parallel.

The arbiter 201 may be coupled to each of the buses shown in Fig. 2. The request and grant lines may be incorporated into the buses as well or may be separately coupled to the arbiter. In fact, the arbiter, rather than being separate as shown, may in fact reside in one or more of the processors or memories, or be distributed among the processors and memories in a manner described further herein.

Assuming a central arbitration scheme utilizing arbiter 201, processor P0 requests transaction T0 for memory M0 from the arbiter, processor P1 requests transactions T0 and T2 for memories M0 and M2, respectively. Processor P2 requests transactions T0, T1 and T2 to memories M0, M1 and M2, respectively. As with the switch embodiment described previously, the arbiter gives the highest priority to processor P0 since that processor has the fewest requests. After P0 is granted its T0 request, P1 has one request pending and P2 has two requests pending, since M0 has already been allocated. Thus, P1 is granted its T2 request since it has higher priority (only one request) and finally P2 is granted its T1 request. Thus, the arbiter grants P0 its T0 request, P1 its T2 request and P2 its T1 request, thus maximizing utilization of buses 200.

An additional factor to be considered in developing an arbitration scheme is to provide fairness in the sense of avoiding starvation for any particular request. Thus, a round robin scheme may be employed in addition to the arbiter prioritizing the requester with the fewest requests first. Various round robin approaches will be described further herein.

A detailed description of an arbiter is provided in the program listing in Appendix A showing procedure arbitrate, which describes one embodiment of an arbiter according to the present invention. There are two main data structures of the interfaces of procedure arbitrate. The first is the Boolean matrix R, which represents the input signals to the arbiter. R[i,j] indicates whether requester i is requesting resource j. The second main data structure is the integer array S, which represents the output values of the arbiter. S[i] indicates which resource is allocated to requester i.

The program listing illustrates the operation of one embodiment of the arbiter each time the arbiter receives requests from the plurality of requesters. As described previously, those requests may be received synchronously. On entry into the arbitration sequence as represented in the procedure arbitrate, the schedule S is initialized (line 16) to show no requests have been granted and the number of requests are calculated for each requester (lines 18-19). Then for each resource available to the requesters, the procedure first determines if the round robin position is asserted. If so, the request is granted. (lines 24-25).

The round robin position in the illustrated embodiment is determined by the index variables I and J along with the loop variable res. For every "schedule cycle", the requester offset I is incremented (lines 47-48). If I rolls over, the resource offset J is incremented. A schedule cycle includes the execution of the code described by the procedure arbitrate, including the round-robin determination and least choice first arbitration. Note that all resources are scheduled during a schedule cycle.

Starting with I=0 and J=0 for the first schedule cycle, the round robin positions in R[i,j] for three requesters [0:2] and three resources [0:2] for four schedule cycles of the embodiment described in Appendix A are as follows:

Schedule cycle 0: [0,0], [1,1], [2,2]
Schedule cycle 1: [1,0], [2,1], [0,2]
Schedule cycle 2: [2,0], [0,1], [1,2]
Schedule cycle 3: [0,1], [1,2], [2,0]

Thus, it can be seen that a different position in array R is designated as the start of the round robin for each schedule cycle. The "round-robin positions" within matrix R form a diagonal during every schedule cycle. The starting position for the diagonal moves downward by one element once every schedule cycle (lines 47, 48). When the index J increments, the starting position moves over to the top of the next column in array R. Note that only one of the elements of the diagonal is guaranteed to be granted (if it is asserted) per schedule cycle. That is the first round robin position in each schedule cycle.

While the arbiter has been described in the software listing of procedure arbitrate to provide a detailed example of operation of one embodiment of the arbiter, the same functionality may preferably be implemented in hardware to increase the speed with which the arbiter can respond to requests as described further herein. In other embodiments, portions of the arbiter may be implemented in hardware and other portions in software depending on the needs of a particular system. Further, the iterations of the sequential calculations of the schedule (lines 21-46), may be implemented with the help of look-ahead techniques. That is, as an alternative to the sequential approach described in Appendix A, a look-ahead technique can instead schedule resources in parallel rather than sequentially. Each resource must know what the other resources will decide. Because that information is deterministic, each resource can calculate the needed information. That parallel approach can speed up determinations.

There are additional embodiments of the arbiter. For example, the round-robin scheme can be varied in different ways. If it is desirable to guarantee that more than one element can be granted, the arbiter could consider all elements of the round robin diagonal before other requests are considered each schedule cycle. That is illustrated in detail in Appendix B.

In another embodiment, the "round-robin positions" cover a row of elements in R rather than a diagonal during each schedule cycle. That is illustrated by the following code segment, which replaces lines 24 and 25 in Appendix A.

```
If R[I,(J+res) mod MaxRes] then
  gnt := I
```

Note that with this embodiment, the requester with the requests covered by the row specified by the round robin positions is guaranteed to be granted one of its requests.

In a similar embodiment, the "round robins positions" cover a column of elements in R rather than a row during each schedule cycle. Note that with this embodiment, the resource requested by requests covered by the column specified by the round robin positions is guaranteed to issue a grant. That is illustrated by the following code segment which replaces lines 24 and 25 in Appendix A.

```
If res = 0 then
begin
   r := 0;
      repeat
         if R[(r+I) mod MaxReq, J] then
            gnt := (r+I) mod MaxReq;
         r := r+1;
      until (r= MaxReq) or (gnt <> -1);
end
```

Yet another possibility is to only consider one "round-robin position" per schedule cycle as shown in the following code segment, which replaces lines 24 and 25 in Appendix A.

```
If (res = 0) and req[I,J] then
  gnt := I
```

In still another starvation avoidance approach, a round robin scheme ages requests. The longer a request waits, the higher a priority the request receives and the sooner it is considered by the round robin scheme. Any of the round-robin schemes described guarantee that every request is periodically considered. That way, starvation can be prevented. Other schemes may be implemented to prevent starvation as well. In fact, any scheme that guarantees that every request is periodically considered, prevents starvation. For example a statistical approach can be used to provide that guarantee. One example of such an approach is a random scheme that provides a probability that an element will be visited within a particular time period.

The main parameter of a round-robin scheme is the period at which elements of R are visited. For the arbiter described in Appendix A, the worst-case period is the total number of requesters times the total number of resources (MaxReq*MaxRes). The period can be shortened if the round-robin scheme can guarantee that more than one element of R (if asserted) is granted in one schedule cycle such as in the approach illustrated in Appendix B. That can be useful if the matrix R is large.

The period of the round-robin scheme also determines the minimum fraction of resource usage that a requester is guaranteed to obtain. For example, if in Figure 3A, R[2,1] is always asserted, requester 2 is guaranteed to get access to resource 1 at least once every 9th schedule cycle.

Referring again to the program listing of the procedure arbitrate in Appendix A, once the round robin position has been tested in lines 24 and 25, if the round robin position does receive a grant, then the procedure assigns the granted resource to the position in S corresponding to the requester (lines 39-45) and adjusts number of requests (NRQ values) appropriately.

If no round robin position is true, i.e. no request is being made for that resource by that requester, then the procedure arbitrate determines for a particular resource, i.e., a particular column in R identified by [(res + J) mod MaxRes], the requester with the fewest outstanding requests and that requester is granted its requested resource (lines 29-36).

If either the round robin position or another position in array R is granted a resource, then the procedure updates array S by writing the number of the resource into the position in S corresponding to the requester (line 40). That is S[i] is written with the resource number, where i represents the requester granted the resource. The requester row in array R for the requester that was granted the resource is modified to indicate that there are no requests in that row, since only one request can be granted per requester (line 41)). The number of requests (NRQ) for all the requesters requesting the granted resource is reduced because requests for a resource already allocated cannot be fulfilled and thus are not counted (lines 43-44). The number of requests (NRQ) for the requester that was granted the resource is set to 0 (line 42). The procedure arbitrate then ends by incrementing I and J appropriately (lines 47-48).

The operation of the embodiment illustrated in procedure arbitrate in Appendix A will now be described with relation to Figs. 3A-3C. Assume that I and J are at [1,0] on entry into procedure arbitrate and R is as shown as shown in Fig. 3A. Array R shows that there are three requesters in the system and three resources. Requester 0 is requesting resource 0, requester 1 is requesting resources 1 and 2 and requester 3 is requesting resources 1, 2 and 3.

The number of requests is shown for each of the requesters: NRQ[0] = 1, NRQ[1] =2 and NRQ[2]=3. The initial round robin position in R evaluated by the arbiter is R [1,0] as a result of the initial value of I and J. Because that position is not asserted (i.e. is false), procedure arbitrate checks if there are any other requests for resource 0, and determines that requester 0 and 2 are requesting resource 0. Because requester 0 has the smallest number of requests (NRQ), the arbiter grants resource 0 to requester 0, as indicated by the X at R[0,0]. Procedure arbitrate zeros out row R[0,x] (made false) and reduces the number of requests for requester 2 by one (NRQ[2]=2) and zeros out the number of requests by requester 0 (NRQ[0]=0) as shown in Fig. 3B.

On the next iteration through the allocation loop as shown in Fig. 3B, with the loop variable res =1, the arbiter evaluates round robin position R[2,1] first. Because that position is asserted, the arbiter grants requester 2 that request as indicated by the X. The row R[2,x] is zeroed out (made false) and the number of requests by requester 1 is reduced by one (NRQ[1]=1) as shown in Fig. 3C. The number of requests by requester 2 is set to 0 (NRQ[2]=0). Fig. 3B illustrates that the requester 0 row was zeroed out after the requester was granted a resource in Fig. 3A. The "X" is still shown at R[0,0] to show that resource was allocated to that requester previously.

In the final allocation loop of this schedule cycle shown in Fig. 3C, with the loop variable res=2, the arbiter first evaluates round robin position R[0,2]. That position is false so the arbiter determines if there are any other requests for that resource. The only request for that resource is from requester 1 which is granted resource 2. Thus, during that schedule cycle, because the requester with the fewest choice (requester 0) was serviced first, high utilization efficiency of the resources could be achieved.

As previously stated, if speed is a consideration, which is likely in many if not most applications, a hardware implementation may be preferred. Referring to Fig. 4, a block diagram of a hardware implementation of the arbiter is illustrated. Assume there are MaxReq inputs to the arbiter. Fig. 4 illustrates hardware 400 associated with the first requester (input 0), which corresponds to requester 0 and hardware 402 associated with the last requester (MaxReq-1). The embodiment illustrated in Fig. 4 operates as follows. Each of the elements of R[i,*], which includes elements 403 and 404, is initialized, where * represents the number of resources. Thus, for example, element 403 contains all requests input by the associated requester 0 for the various resources. In addition, the array S, which includes elements 409 and 411 of the illustrated hardware, is reset. The requests for the illustrated elements 403 and 404 are summed in summers 405, 406 to generate a number of requests (NRQs) 407 and 408 for the respective requesters. If R[I+res, J+res] = 1, then grant (GNT) 410 is set to (I+res) through multiplexer 412. That is, the round robin position is granted. Otherwise, grant (GNT) 410 is set to the input with the minimum number of requests, which is determined in minimize circuit 414. If there is no request for the current output, grant is set to -1 (note that the logic to set grant to -1 is assumed to be contained within grant (GNT) 410). If grant (GNT) 410 is not -1, then the location S[gnt] (e.g. 409) is set to J+res, which indicates the resource granted to that requester. Each of the elements of R[gnt, *] is set to 0, where * represents the resources. The register 416 holding "res" is incremented so res = res + 1. The hardware continues the calculations for S until res = MaxRes, that is, the last resource is scheduled when S[res=MaxRes-1] is evaluated. The register 418 containing the index I is incremented so I = I+1. If I = MaxReq, then I is set to 0 and the register 420 containing J is incremented. The embodiment illustrated in Fig. 4 is an illustrative block diagram and does not show all the details described. As would be known to those of skill in the art, those details along with many other hardware implementations can be provided to implement the various embodiments described herein.

As is apparent from the software descriptions provided, the arithmetic operations are done mod MaxReq and mod MaxRes. Assuming that MaxReq and MaxRes are powers of 2, the mod operations are simply implemented by having fixed number of register bits (MaxReq and MaxRes, respectively).

In another embodiment the calculation of priorities may be varied. For example, rather than assigning priorities to the requesters as illustrated in procedure arbitrate, priorities can be assigned to the resources. More specifically, the priority of a resource is given by the number of requesters that request it. That is, the highest priority is given to the resource that is requested by the fewest number of requesters, and the lowest priority is given to the resource that is requested by the most number of requesters. Resources are assigned to the requesters one after the other.

Thus, assuming a configuration as shown in Figs. 5A-5C, with three requesters and three resources, each requester 0, 1 and 2 is evaluated in turn to see what resources are being requested and the requested resource with the highest priority is granted to that request. Assume the same round robin positions and I and J values as in Figs. 3A-3C. "Req" represents the loop variable that together with offset I, determines the requester being scheduled. In this embodiment, the arbiter is modified to determine priorities with relation to resources and not requesters.

The number of requests (NRQ) for each of the resources initially is: NRQ[0] = 1, NRQ[1] =2 and NRQ[2]=3. Assume the arbiter of this embodiment uses a similar round robin scheme as that used by the arbiter in Figs. 3A-3B. In Fig. 5A, the arbiter evaluates the initial round robin position R [1,0], which results from the initial value of I and J. Because that position is not asserted (i.e. is false), the arbiter according to this embodiment checks if there are any other requests by requester 1. Because requester 1 is requesting resources 1 and 2 and resource 1 has a higher priority than resource 2, the arbiter grants resource 1 to requester 1. All the requests for resource 1 are zeroed out as shown in Fig. 5B.

On the subsequent iteration to assign resources illustrated in Fig. 5B, the arbiter evaluates the round robin position R[2,1] first. Because that position was zeroed out, the arbiter checks if requester 2 is making any other requests for resources. Requester 2 is also requesting resource 2, its only remaining request and thus the arbiter grants resource 2 to requester 2. The column representing resource 2 is zeroed out. Referring to Fig. 5C, on the final iteration which completes checking all requests in R, the arbiter evaluates round robin position R[0,2] first. Since that position is false, the arbiter then determines if any other resources are being requested by requester 0. Since requester 0 is also requesting resource 0, the arbiter grants that resource to requester 0.

In still another embodiment, the arbiter calculates priorities for both the requesters and resources, and uses those priorities to calculate a priority for each element in R. For example, the number of requests for a particular resource may be added with a number of requests made by a requester, to create each position in R. The scheduler then iteratively grants the request with the highest priority. That way, the priority of an element in R indicates the degree of freedom of its requester as well as of the requested resource. As in other embodiments, a scheme to prevent starvation may be included. However, a scheme to prevent starvation may not always be necessary. For example, some applications may have a guaranteed pattern of requests that makes the need for starvation avoidance unnecessary.

The arbiters described so far have been for a centralized arbiter implementation. A distributed version of the least choice first arbiter is also possible. That version is distributed in that the selection of requests is performed by arbitration logic associated with the resources and requesters rather than by centralized arbiter logic. For example, assume a transport mechanism such as the simple 3 X 3 switch shown in Fig. 6 having three input ports and three output ports. Thus, there are three requesters and three resources. Node 601, associated with input port 0, has one request 602 for output port 0. Node 603, associated with input port 1, has two requests 604 for output ports 1 and 2, respectively. Node 605 associated with input port 2 had three requests 606, for output ports 0, 1 and 2, respectively. As shown in Fig. 6, output port 0 has requests from input port 0 and input port 2. Output port 1 has requests from input port 1 and input port 2. Output port 2 has requests from input port 1 and input port 2.

Referring to Fig. 7, showing an exemplary distributed implementation for the arbitration scheme disclosed herein, the requesters REQ 0, REQ 1 and REQ 2 supply request signals to and receive grant signals from each of the resources RES 0, RES 1 and RES 2. For example, requester REQ 0 sends request signal 701, 703 and 705 to resources RES 0, RES 1 and RES 2, respectively. Requester REQ 0 receives grant signals 707, 709 and 711 from resources RES 0, RES 1 and RES 2. In one embodiment, during an arbitration period, each resource receives request vectors indicating which requests are asserted from all requesters. A resource grants one of the received requests according to the priority scheme described herein and notifies that requester via an asserted grant signal.

The request vectors may be provided as shown in Fig. 8 where request vector 0 corresponds to input port 0, request vector 1 to input port 1 and request vector 2 to input port 2. Thus, the request vectors indicate by an asserted bit, which output port is requested. The resources can determine from the request vectors how many requests a particular requester has made. From the example shown in Figs. 6 - 8, requester REQ 0 (input port 0) has made 1 request for output port 0, requester REQ 1 (input port 1) has made 2 requests for output ports 1 and 2, and requester REQ 2 (input port 2) has made 3 requests for output ports 0, 1 and 2.

In one embodiment, a sequential scheme is utilized to implement a distributed arbiter. The sequential scheme in that embodiment requires n rounds with n being the number of resources. For each round i (i = 0..n-1):

(1) Each requester that has not yet received a grant and that is requesting resource i sends a request. The request is accompanied by a priority which is the inverse of the number of resources that the requester is requesting and that have not yet been scheduled in a previous round. The priority indication may be provided as the number of requests being made by the particular requester.

(2) The resource selects the request with the highest priority (the requester making the fewest requests) and sends a grant to the corresponding requester. If there is more than one request with the highest priority, a round-robin scheme is used to select one of the requesters. Note that all the requesters are informed by all resources when a resource has been granted so requesters know what has been scheduled in a previous round.

The rounds continue until all the resources have been scheduled.

A second embodiment uses an iterative scheme with requesters and resources making choices in parallel. In one iteration of the parallel approach according to this embodiment:

(1) Each requester that has not yet accepted a grant, sends a request to each resource that it is requesting and that has not yet been allocated. The request is accompanied by a priority which is the inverse of the number of requests the requester is sending.

(2) Each resource selects the request with the highest priority. If there is more than one request with the highest priority, a scheme such as a round robin scheme may be used to select one of the requests. A grant is sent to the requester of the selected request. The grant is accompanied by a priority which is the inverse of the number of requests the resource has received.

(3) If a requester receives more than one grant it accepts the grant with the highest priority. If there is more than one grant with the highest priority a round robin scheme is used to select the grant to be accepted. If a grant is accepted, the corresponding resource is allocated.

While in the worst case n iterations are needed to schedule all resources, a smaller number of iterations will generate schedules which are nearly optimal for most patterns of requests.

Note that either or both of the resources and requesters in a distributed approach may still implement a round robin scheme to ensure that no starvation occurs in the system. Further, the distributed arbiter may implement only the least choice priority scheme on either the resource side or the requester side.

The distributed approach described herein for the switch in Fig. 6 is equally applicable to systems requiring arbitration for resources such as the multiprocessor system illustrated in Fig. 2.

Referring to Figs. 9A-9D and the program listing in Appendix C, which describes the operation of one embodiment of a sequential distributed arbiter, array R has a 1 in each position R[i,j] where requester i is requesting resource j. Assume that I,J=[1,0] initially. That results in an initial round robin position of R[1,0], which is shown in Fig. 9A. Figs. 9A-9D show the resource RES and the number of requests for each requester shown as (requester:requests). As shown in procedure arbitrate in Appendix C, for each resource rs, procedure resource is called, which checks to see if the round robin position is asserted and if not, finds the requester with the highest priority. After a resource is allocated, procedure requester is called for each requester rq to recalculate the number of requests. Referring to Fig. 9A again, in which the arbitration for resource 0 is shown, the round robin position R[1,0] happens to be asserted and thus resource 0 is granted to requester 1. The circled position in array R indicates the round robin position. The "X" indicates the granted request. After resource 0 is allocated, the requests for requester 1 are zeroed out since that requester has been granted a resource, and the requesters recalculate their number of requests and resend their requests to the resources. The initial number of requests being made by the various requesters is also shown in Fig. 9A. In the embodiment illustrated in Appendix C and Figs. 9A-9D, requests for a scheduled resource are no longer considered when recalculating the number of requests. That implies that grants are known by all requesters. In other embodiments, scheduled resources may be considered in recalculating the number of requests, when, e.g., grants are only known by the requester granted the resource. In other embodiments, it is also possible not to recalculate the number of requests at all.

While the requesters, in this embodiment, provide the number of requests as an indication of their priority, in another embodiment, the requesters may send a number directly indicative of their priority.

After the arbitration for resource 0 is completed the distributed arbiter arbitrates the requests for resource 1 as illustrated in Fig. 9B. The round robin position is incremented to be R[2,1] as indicated by the circle in that position in array R. Since that position is false, the sequential distributed arbiter goes on to evaluate the other requests for resource 1. Requester 3 has the highest priority for resource 1 (NRQ=1 as opposed to NRQ=2 for requester 0) and is therefore granted its request as indicated by the X at R[3,1].

After resource 1 is allocated, the requests for requester 3 are zeroed out since that requester has been granted a resource (resource 1) and the requesters recalculate their number of requests and resend their requests to the resources. The resulting calculations are shown in Fig. 9C.

Resource 2 is now allocated as shown in Fig. 9C. The round robin position is incremented to be R[3,2] as indicated by the circle in that position in array R. Since that position is false, the sequential distributed arbiter goes on to evaluate the other requests for resource 2. Requesters 0 and 2 have requests for resource 2. Since requester 0 has only one request as compared to two requests for requester 2, requester 0 is granted resource 2 as indicated by the X at R[0,2].

After resource 2 is allocated, the requests of requester 0 are zeroed out since that requester has been granted a resource (resource 2) and the requesters recalculate their number of requests and resend their requests to the resources. The resulting calculations are shown in Fig. 9D.

Resource 3 is now allocated as shown. The round robin position is incremented to be R[0,3] as indicated by the circle in that position in array R. Since that position is false, the sequential distributed arbiter goes on to evaluate the other requests for resource 3. Since requester 2 is the only requester, requester 2 is granted resource 3 as indicated by the X at R[2,3]. That completes an arbitration cycle for the resources. All the resources have been granted according to priority of the requesters or the round robin mechanism. Note that because the resources were allocated sequentially, none of the requesters received multiple grants during the arbitration cycle for the resources. While the particular implementation described in Appendix C and illustrated in Figs. 9A-9D, have used a round robin scheme, any starvation avoidance scheme (or none at all) may be utilized depending on the system requirements.

Fig. 10 provides another view of a system utilizing a sequential distributed arbiter and illustrates the information flow between resources and requesters in an embodiment of a sequential distributed arbiter. Fig. 10 shows three requesters and three resources. In the illustrated embodiment, each requester (RQ0, RQ1, RQ2) sends to each resource (RS0, RS1, RS2) its requests R and its number of requests NRQ. For example, RQ0 sends requests R[0,0], R[0,1] and R[0,2] together with its number of requests NRQ [0] to resources RS0, RS1 and RS2, respectively. In one sequential arbiter embodiment, RS0 grants requests prior to RS1, which grants its requests prior to RS2. In that way, requests from requesters already granted a resource can be eliminated during sequential operations, thus ensuring that multiple grants are not supplied to requesters. In other sequential distributed arbiters, the order in which resources are allocated may be varied as explained further herein.

The number of requests function as the priority indication. Other embodiments may encode a priority indication differently. For example, the inverse of the number of requests may be sent.

The number of wires utilized in the embodiment shown in Fig. 10, assuming a separate wire is used for each transmission, is $n^2(\lceil \log_2(n+1) \rceil +2)$, where n represents the number of resources (and assuming the system is symmetric in terms of the number of requesters and the number of resources). NRQ has values from 0..n; therefore, the number of values is n+1. Since NRQ only needs to be defined for R[i,j]=1, in which case it has values 1..n, NRQ could be encoded such that the numbers of wires is $n^2(\lceil \log_2 n \rceil +2)$, where $\lceil x \rceil$ is defined as a ceiling function, e.g., $\lceil \log_2 7 \rceil =3$. Other implementations may use $\lceil \log_2(n+1) \rceil$ to encode NRQ. Thus, where n=3, the number of wires utilized is 36 for either implementation. That assumes that the request and grant utilize one wire each and the binary encoding of NRQ utilizes 2 wires. Thus, four wires connect each requester to each resource in the embodiment illustrated in Fig. 10.

There are many other ways to couple the resources and the requesters. Additional exemplary embodiments will be described. Generally, however, there is a trade off between space and time in designing the data paths that connect the requesters and the resources. If every signal has its own wire, and, as a result, the data path is not time multiplexed in any manner, arbitration time can be reduced to approximately the time taken by the resource to calculate the grants. For a sequential distributed arbiter, where every signal has its own wire, the time t to complete an arbitration cycle is a function of the size of the system, that is it is n times the time taken to calculate the grants for any one resource, where n is the number of resources. For a parallel distributed arbiter, which is discussed further herein, the time to complete an arbitration varies according to the number of arbitration iterations necessary to allocate the resources. As the number of wires are reduced, and the degree of multiplexing increases, the total arbitration time increases in both sequential and distributed arbiters. Various systems may make different space-time tradeoffs according to system requirements.

Referring to Figs. 11A and 11B and the software listing in Appendix D, operation of a parallel distributed arbiter will now be described. In a parallel distributed arbiter, each resource calculates its grants in parallel with other resources.

An exemplary embodiment of a parallel distributed arbiter is described in Appendix D. Before an arbitration iteration, in which resources determine which requests to grant and requesters determine which grant to accept, the resources initialize their grants in procedure init_resource. All the requesters calculate their number of requests, which provides an indication of their priorities in procedure init_requester. A first iteration of the operation of the parallel distributed arbiter that is modeled in Appendix D, is provided in Fig. 11A. Fig. 11A shows the iteration number IT, the number of requests (NRQ) for each requester, which resource was granted to which requester in GNT, shown as (resource:requester), the number of requests for a resource (NRS) shown as (resource:requests) and finally the accepted grants (ACC) shown as (requester:resource). All the resources determine their grants simultaneously. That is modeled in the listing in Appendix D by calling procedure resource for each resource in parallel without the resources coordinating the granted requests and without any of the requesters recalculating their NRQs as was done in the sequential distributed arbiter illustrated in Appendix C.

The round robin mechanism used in Appendix D is similar to the ones described previously. Arbitration for each of the resources will be described with relation to Fig. 11A for iteration 0. Note that it is preferable, although not necessary, that the resources make their arbitration decisions in parallel.

Resource 0 sees two requests (NRS 0:2). However, since the round robin position indicated by a circle at position R[1,0] indicates a request, resource 0 grants requester 1 its request (GNT 0:1) as indicated by the square at position R[1,0]. There is no need to evaluate requests based on priority.

Note that the round robin positions are coordinated among the requesters as well as the resources in the embodiment shown. In other embodiments, the round robin positions are independently determined in each requester and resource. Other starvation avoidance mechanisms, e.g., a random selection prior to allocating a resource to requesters based on priority, may also be used in a parallel distributed arbiter. In another embodiment, a parallel distributed arbiter uses no starvation avoidance mechanism at all.

For resource 1, the round robin position is R[2,1], which does not indicate a request. Therefore, resource 1 arbitrates requests based on priority as described in procedure resource in Appendix D. Resource 1 has two requests (NRS 1:2), from requester 1 and requester 3. Since requester 1 has four requests (NRQ 1:4) and requester 3 has two requests (NRQ 3:2), requester 3 has a higher priority and resource 1 grants requester 3 its request (GNT 1:3) as indicated by the square at position R[3,1].

For resource 2, the round robin position is R[3,2], which does not indicate a request. Therefore, resource 2 arbitrates requests based on priority as illustrated in procedure resource. Resource 2 has three requests (NRS 2:3), from requesters 0, 1 and 2. Requester 0 has one request (NRQ 0:1); requester 1 has four requests (NRQ 1:4); and requester 2 has three requests (NRQ 2:3). Resource 2 grants requester 0 its request (GNT 2:0) since it has a higher priority (lower NRQ) than the other requesters.

For resource 3, the round robin position is R[0,3], which does not indicate a request. Therefore, resource 3 arbitrates requests based on priority as described in procedure resource. Resource 3 has three requests (NRS 3:3), from requesters 1, 2 and 3. Requester 1 has four requests (NRQ 1:4); requester 2 has three requests (NRQ 2:3); and requester 3 has two requests (NRQ 3:2). Resource 3 grants requester 3 its request (GNT 3:3) since requester 3 has a higher priority (lower NRQ) than the other requesters.

Note that requester 2 was not granted any of its requests during iteration 0. Although no ties were illustrated in the examples given, in the event of a tie, many approaches can be used to break the tie, such as a round robin or random scheme.

Unlike the sequential arbiter, it is possible for requesters to receive multiple grants. That can be seen in Fig. 11A where requester 3 receives grants from resource 1 and from resource 3. Resource 3 therefore has to choose between two grants. That can be accomplished through a variety of mechanisms. A round robin scheme could be utilized. A random scheme could be employed. A priority scheme could also be utilized, alone or in conjunction with a starvation avoidance mechanism.

In the embodiment illustrated in Fig. 11A and described in appendix D, a round robin scheme is used in conjunction with an arbitration scheme accepting a grant based on the number of requests that were made for a particular resource. The requester selects as the highest priority grant, the grant from a resource with the fewest requests and thus accepts the grant from that resource.

The operation of such an embodiment is described in Appendix D and illustrated in Fig. 11A. Preferably, if time is important, the requesters determine their acceptances in parallel. Since requester 0 and 1 received only one grant each, they accept their respective grants. Since requester 2 has no grants to accept, it does not accept a grant. Because requester 3 received two grants, as indicated by the squares in R[3,1] and R[3,3]. Requester 3 uses an arbitration approach to decide which grant to accept. The arbitration approach selects as the highest priority grant, the grant coming from the resource with the fewest requests. Each of the resources, supplies to the requesters the number of requests received by the resource (NRS), along with a grant indication. Another indication of priority of the resources could also be supplied, e.g., the inverse of the number of requests. From Fig. 11A, it can be seen that resource 1 received two requests (NRS 1:2) and resource 3 received three requests (NRS 3:3). Thus, resource 1 has the highest priority and is accepted by requester 3. Requester 3 sends back an accept indication to resource 1 as indicated by the X at R[3,1]. As illustrated in the program listing of procedure requester in Appendix D, each requester can also employ a starvation avoidance mechanism, such as round robin, prior to evaluating the grants based on priority. That ensures that a particular grant is guaranteed to be accepted within a predetermined amount of time if it is continually asserted. In one embodiment, all of the requesters that accepted a grant, clear their NRQs after the grants are accepted. That completes the first iteration of the parallel distributed arbiter. Note that only three of the four resources have had grants accepted.

A second iteration is illustrated in Fig. 11B. Because all of the requesters that accepted a grant in the previous iteration have cleared their NRQs, only requester 2 has a non-zero NRQ. Since resource 3 is the only resource whose grant was not accepted in the previous iteration, it is the only resource in procedure resource that evaluates requests again. The round robin position does not have to change during the iterations and in the operation of the distributed arbiter illustrated in Appendix D, the round robin position does not change. Because the round robin position is not asserted, resource 3 goes on to evaluate outstanding requests. As requester 2 is the only request that it has (R[2,3]), resource 3 supplies a grant to requester 2 (GNT 3:2) along with the number of requests resource two received. In this case, that number is 1 (NRS 3:1).

Requester 2 is the only requester that has not received a grant, and accepts the grant from resource 3. That completes the second iteration.

Fig. 12 provides another view of a system utilizing a parallel distributed arbiter and illustrates the information flow between resources and requesters in an embodiment of a parallel distributed arbiter. To simplify the figure, Fig. 12 shows only two requesters and two resources. In the illustrated embodiment, each requester (RQ0, RQ1) sends to each resource (RS0, RS1) its requests R and its number of requests NRQ. For example, RQ0 sends requests R[0,0] and R[0,1] to resources RS0 and RS1, respectively and NRQ [0] to both. In a parallel arbiter embodiment, the resources preferably evaluate their requests and provide their grants in parallel. Thus, a resource determines which request to grant generally independent of the other resources (the round robin positions may be coordinated). For that reason, a requester with multiple requests may get multiple grants.

The number of requests, NRQ, sent by each requester functions as the priority indication. Other embodiments may encode a priority indication differently. For example, the inverse of the number of requests may be sent. In another alternative, the requester may send a vector of requests as shown in Fig. 8, which each resource receives and from which each resource independently calculates a priority based on the number of requests.

In addition to sending grants, such as GNT[0], as did the sequential arbiter, each resource also sends its priority in the form of the number of requests it received, shown as NRS in Fig. 12. Again, the priority may be encoded as a number of requests, an actual priority, provided as a vector of requests or in any other manner suitable to inform the requester about the priority of the resource, if resource priority is used to determine which grant to accept. Finally, each requester sends an acceptance (ACC) back to resource.

There is more information transferred between the requesters and resources in the illustrated embodiment of the parallel distributed arbiter as compared to the sequential distributed arbiter illustrated in Fig. 10. Therefore, more wires are required in an embodiment illustrated in Fig. 12, assuming a separate wire is used for each transmission. For each connection providing information from a requester rq to a resource rs, the request (R[rq,rs]) requires one wire; the binary encoded number of requests (NRQ) requires $\lceil \log_2(n+1) \rceil$ wires, where n is the number of resources; the accept (ACC) requires one wire. For each connection providing information from a resource to a requester, the grant GNT requires one wire; and the binary encoded number of requests received by a resource (NRS) requires $\lceil \log_2(n+1) \rceil$ wires, where n is the number of requesters. For a system with n requesters and n resources, the total number of wires utilized is $n^2(2\lceil \log_2(n+1) \rceil + 3)$.

There are many other ways to couple resources and requesters. Additional exemplary embodiments will be described. As previously stated, however, there is a trade off between space and time in designing the data paths that connect the requesters and the resources. If every signal has its own wire, and, as a result, the data path is not time multiplexed in any manner, arbitration time can be reduced to approximately the time taken by the resource to calculate the grants. For a parallel distributed arbiter, the time to complete an arbitration cycle varies according to the number of arbitration iterations taken to allocate the resources. Since a small number of iterations is sufficient to generate a nearly optimal schedule, the parallel distributed arbiter is potentially faster than the sequential distributed arbiter, particularly if the number of requesters and resources is large. As the number of wires are reduced, and the degree of multiplexing increases, the total arbitration time increases. Various systems may make different space-time tradeoffs according to system requirements.

In another embodiment of an arbitration system, rather than a switch model, generally illustrated in Figs. 10 and 12, a bus based distributed arbiter may be implemented as shown in Fig. 13. Each requester RQ_0, RQ_1 and RQ_2 is coupled to the resources RS_0, RS_1 and RS_2 through a bus 131 of width b.

Operation of a sequential bus based distributed arbiter is illustrated in Fig. 14. Requester RQ_0 places both its request for resource 0 R[0,0] and its priority (in terms of number of requests) NRQ[0] on bus 131. As shown in Fig. 14, that takes one cycle. Note the term cycle does not necessarily refer to a single bus clock. Rather, the term is intended to more generically refer to the amount of bus time that it takes to transfer a unit of information. On some buses that may be the amount of time to transfer a word across the bus. On other buses, the overhead for the information transfer involves more bus cycles. Assume that bus 131 is of a sufficient width to transport both NRQ ($\lceil \log_2(n+1) \rceil$) and requests (1 bit) in one "cycle". That assumes that the bus width $b = \lceil \log_2(n+1) \rceil + 1$. during each subsequent cycle another requester rq places its request R[rq,0] for resource 0 and its priority (NRQ[rq]) on the bus. Once a resource has received all its requests, it grants one of the requests according to a least choice first approach in which requesters having the fewest requests are granted their requests first. In addition, a round robin scheme or other starvation avoidance scheme, as described herein, may be utilized. Thus, it takes the requesters, assuming n requesters, n cycles to place their information on the bus. The grant takes one cycle. The time to compute the grant is not shown. For each resource, the arbitration time, in terms of bus cycles is n+1 cycles. The arbitration time t for n resources is t = n X (n+1), which = $n^2 + n$. If n = 16, then t = 272. If n=32, then t = 1056. Because the arbitration time t is based on n^2, for large bus based systems, the sequential distributed arbiter may consume a lot of cycles.

A bus based system may also use a parallel distributed arbiter. Operation of a parallel distributed arbiter is illustrated in Fig. 15, which shows one iteration of a parallel distributed arbiter for n requesters and n resources. Each requester rq, sequentially sends out a vector R[rq] of requests indicating the resources it is requesting for the iteration and its number of requests NRQ[rq] during 1501. Assume the bus is wide enough to simultaneously supply the vector of requests and the number of requests from one requester. That requires $\lceil \log_2(n+1) \rceil$ wires for NRQ and n wires for the vector of requests. For 32 resources, that could take 32 vector bits and 6 bits for NRQs. For n requesters, that takes n cycles. If the bus is smaller, multiple bus cycles are needed for each requester. Each resource sends its grant and its number of received requests (NRS) during time period 1503 on the bus. Each resource consumes one cycle. For n resources, that takes n cycles. Again, the bus is assumed to be wide enough to support sending its grants (GNT, $\log_2(n)$ wires) along with the number of received requests (NRS, $\lceil \log_2(n+1) \rceil$ wires). If the bus is smaller, it will take extra bus cycles to transfer the information. Once the grants are received, each requester sends its accepts (ACC, $\log_2 n$ wires) on the bus during 1505. For n requesters that takes n cycles. So the total number of cycles to accomplish an iteration of the distributed arbiter operation illustrated in Fig. 15 is (3 X n). If n = 32 and the number of iterations required is 4, the arbitration cycle takes 384 cycles as compared to 1056 for the sequential distributed arbiter. That is a significant improvement over the sequential distributed arbiter. Thus, depending on the circumstances of a particular implementation, for large networks or where large numbers of requesters and resources are arbitrating, a parallel distributed arbiter may provide advantages in terms of time savings.

Referring to Figs. 16A and 16B, another example of operation of a parallel distributed arbiter is shown. Referring to Fig 16A, requesters 160, 161, 162 and 163 and are requesting resources 164, 165, 166 and 167. Requester 160 has one request for resource 166. Requester 161 has three requests, for resources 165, 166 and 167. Requester 162 has three requests for resource 164, 166 and 167. Requester 163 has two requests, for resource 165 and resource 167.

Referring to Fig. 16B, the resources grant the requests as follows. Resource 164 provides a grant to its single requester 162. For resource 165, requester 163 has fewer requests than requester 161 and therefore the highest priority. Therefore, resource 165 grants requester 163 its request. Resource 166 grants requester 160, with only one request, its request. Finally, resource 167 grants requester 163 its request, since it was the requester with the fewest requests. Thus, at the end of the grant phase shown in Fig. 16B, requesters 160 and 162 have received one grant and requester 163 has received two grants.

During the accept phase shown in Fig. 16C, requesters 160 and 162 send accepts in response to their one received grant. Requester 163 accepts the grant from resource 165 since resource 165 had received two requests as opposed to the three requests received by resource 167, and thus resource 165 had the highest priority. At the end of the first iteration, all the requesters except 161 have accepted grants.

The second iteration is shown in Figs. 16D-16F, where requester 161 resends its three requests in Fig. 16D. Fig, 16E shows the grant sent by resource 167 to requester 161 (along with the previously sent grants). Fig. 16F shows the accept sent from requester 161 to resource 167 (along with the previously sent accepts). Thus, two iterations have completely allocated all the resources. In general, two to four iterations are sufficient to provide close to optimal allocation of resources. For n=32, four iterations has provided good results. Note that a starvation avoidance mechanism was not illustrated in Figs. 16A-16F.

While the number of iterations during each arbitration cycle may be fixed in certain embodiments, in other embodiments, the number of iterations may be varied dynamically to respond to such factors as an estimation of how optimal the scheduling is, or loading conditions. For example, if a requester receives large numbers of grants, that may suggest sub-optimal scheduling since only one grant is accepted leaving the resources with the other grants unallocated. On the other hand, if resources stop changing grants, no more iterations are needed.

There are numerous variations that can be made to the embodiments described herein. For example, the transmission of the grants can be omitted if each requester is able to calculate all grant values. That implies that the resources and requesters are synchronized in regards to round robin positions. Assume that each requester is also a resource and thus forms a requester/resource pair. For example, if a requester is assumed to be a node on a network, that node is also likely to be a resource. For each resource or requester to calculate all grant values, it means that each requester/resource pair receives all requests, which may take the form of the request vectors illustrated in Fig. 8 or shown operationally in Fig. 15.

Other variations are possible. For example, the transmission of the NRQ[0..n-1] may be omitted if the resources are able to calculate those values based on the requests sent by the requesters. For example, if the requesters transmit the requests in n cycles, R[0, 0..n-1] in the first cycle, R[1, 0..n-1] in the second cycle, etc. The resources can, from that information, calculate NRQ and subsequently GNT and that reduces the transmission time from 3n to n cycles. Note that the variations tend to increase state and complexity of the resources.

In another embodiment the sequential distributed arbiter, rather than scheduling the resources in a fixed order, uses NRS[0..n-1] to determine the order. That is, the resources are scheduled in order of priority.

While the embodiments have indicated the number of requester requests (NRQ) and number of received requests by a resource (NRS) are updated during the arbitration cycle, it is also possible to not update those values. That is true for both parallel and sequential distributed arbiter embodiments.

Referring to Fig. 17, in another implementation, an open collector bus structure 1701 is used to calculate the minimum of NRQ[0], ..., NRQ[n-1], that is, to determine the requesters with the highest priority. In this implementation the requester/resource pairs 1703-1705 (RQ_0/RS_0 through RQ_{n-1}/RS_{n-1}), encode NRQ as a unary encoded message. All of the requesters send their NRQs simultaneously. The open collector bus 1701 sees the wired-NOR of all the NRQs. The result of the wired-NOR is the inverted value of the highest priority requester. For example, assume requester 1703 has priority 4. That is encoded onto bus 1701 as "0...01111". Assume another requester has priority 2, which is encoded onto bus 1701 as "0...00011". Assume those are the only two requesters with requests. The wired-NOR of those two values is "1...10000". One way to use this information is to have requesters with lower priority remove their requests and have the resource choose one of the remaining "highest priority" requests to grant. That is, resources only receive requests from the highest priority requesters. That way, there is no need to determine which requester is highest priority.

As would be readily apparent to those of skill in the art, there are many ways to modify the bus structure of 1701. For example, each requester may put the inverse of its NRQs on the bus and wire-NOR those together. For example, assume requester 1703 has four requests (NRQ[0] = 4). That is encoded onto bus 1701 as "1...10000". Assume another requester has two requests (NRQ[n] = "1...11100". Assume those are the only two requesters with requests. The wired-NOR of those two values is "0...00011", which represents a requester with two requests, the highest priority requester. The requesters see that any requester with more than two requests should withhold requests for the resource to be allocated.

After the open collector bus "arbitration", the requester(s) having the highest priority send their requests to the resource being scheduled. The resource knows that all requests received have equal priority and therefore the resource grants a request according to a round robin, random or other scheme to prevent starvation if there is more than one requester (which necessarily have the same priority).

Thus, an arbiter has been described that achieves high aggregate usage of the resources while still providing a minimum level of fairness, mainly in the sense that starvation of individual requests is prevented. The arbiter makes decisions based on granting resources according to a number of requests made by a requester, number of requests made for a resource or a combination of the two. The approach described herein can be utilized in a centralized arbitration scheme or in a distributed arbitration scheme. Referring to Figs. 18A and 18B, those two approaches are respectively illustrated again in block diagram form.

There are several differences between the centralized and distributed arbitration approaches described herein. Typically, each of the requesters and resources in both the centralized and distributed arbitration schemes, where the arbiter is being utilized to allocate ports of a switching network, are on line cards or network interface cards. The line cards are coupled to the switch fabric 1801. However, in the centralized arbitration approach, the centralized arbiter is typically included on the board that includes the switch fabric. In the distributed approach, the distributed arbiters are located on the various line or network interface cards. Thus, a distributed approach provides the advantage of modularization over the centralized approach. That decentralization provides for increased fault tolerance, as a failure in an arbiter would not tend to bring the whole switch down. In addition, for a very large switch, e.g., 128 ports or greater, wiring problems due to the large number of ports may also make a centralized approach undesirable. Further, with a large switch, the parallel distributed approach has advantages over the sequential distributed arbiter since the arbitration time is proportional to the number of iterations (where the number of iterations typically is $\log_2$(number of resources)) rather than the number of resources.

The description of the invention set forth herein is illustrative, and is not intended to limit the scope of the invention as set forth in the following claims. The various embodiments may be used in various combinations with each other not particularly described. For instance, various of the starvation avoidance approaches described herein may be used in the distributed scheme. Note also that the sequential approach described herein can also be utilized by a centralized distributed arbiter. Further, the software described herein is used solely to describe operation of the system and many other data structures and hardware and software solutions can be implemented based on the teachings herein. Those and other variations and modifications of the embodiments disclosed herein, may be made based on the description set forth herein, without departing from the scope and spirit of the invention as set forth in the following claims.

APPENDIX A

```
var
 I: 0..MaxReq-1;                                  (* current round-robin requester offset *)
 J: 0..MaxRes-1;                                  (* current round-robin resource offset *)
 req: 0..MaxReq-1;                                (* requester *)
 res, r: 0..MaxRes-1;                             (* resource *)
 gnt: -1..MaxReq-1;                               (* granted request *)
 min: 0..MaxRes+1;                                (* minimum number of requests *)
 R: array [0..MaxReq-1, 0..MaxRes-1] of boolean;  (* R[i,j] says whether requester i is requesting resource j *)
 nrq: array [0..MaxReq-1] of 0..MaxRes;           (* number of resources requested by a requester *)
 S: array [0..MaxReq-1] of -1..MaxRes;            (* S[i] contains the granted request for requester i;
                                                      if it contains -1, no request was granted *)
procedure arbitrate;
begin
 for req := 0 to MaxReq-1 do
 begin
  S[req] := -1;                                   (* initialize schedule *)
  nrq[req] := 0;
  for res := 0 to MaxRes-1 do
   if R[req,res] then nrq[req] := nrq[req]+1;     (* calculate number of requests for each requester *)
 end;
 for res := 0 to MaxRes-1 do                      (* allocate resources one after the other *)
 begin
  gnt := -1;
  if R[(I+res) mod MaxReq,(J+res) mod MaxRes] then(* round-robin position wins *)
   gnt := (I+res) mod MaxReq
  else                                            (* find requester with smallest number of requests *)
  begin
   min := MaxRes+1;
   for req := 0 to MaxReq-1 do
   begin
    if (R[(req+I+res) mod MaxReq,(res+J) mod MaxRes]) and (nrq[(req+I+res) mod MaxReq] < min) then
    begin
     gnt := (req+I+res) mod MaxReq;
     min := nrq[(req+I+res) mod MaxReq];
    end;
   end;
  end;
  if gnt <> -1 then
  begin
   S[gnt] := (res+J) mod MaxRes;
   for r := 0 to MaxRes-1 do R[gnt, r] := false;
   nrq[gnt] := 0;
   for req := 0 to MaxReq-1 do
    if R[req, (res+J) mod MaxRes] then nrq[req] := nrq[req]-1;
  end;
 end;
 I := (I+1) mod MaxReq;
 if I = 0 then J := (J+1) mod MaxRes;
end;
```

APPENDIX B

```
var
 I: 0..MaxReq-1;                                   (* current round-robin requester offset *)
 J: 0..MaxRes-1;                                   (* current round-robin resource offset *)
 req: 0..MaxReq-1;                                 (* requester *)
 res, r: 0..MaxRes-1;                              (* resource *)
 gnt: -1..MaxReq-1;                                (* granted request *)
 min: 0..MaxRes+1;                                 (* minimum number of requests *)
 R: array [0..MaxReq-1, 0..MaxRes-1] of boolean;   (* R[i,j] says whether requester i is requesting resource j *)
 nrq: array [0..MaxReq-1] of 0..MaxRes;            (* number of resources requested by a requester *)
 S: array [0..MaxReq-1] of -1..MaxRes;             (* S[i] contains the granted request for requester i;
                                                            if it contains -1, no request was granted *)
procedure arbitrate;
begin
 for req := 0 to MaxReq-1 do
 begin
  S[req] := -1;                                    (* initialize schedule *)
  nrq[req] := 0;
  for res := 0 to MaxRes-1 do                      (* calculate number of requests for each requester *)
   if R[req,res] then nrq[req] := nrq[req]+1;
 end;
 for res := 0 to MaxRes-1 do                       (* check round-robin positions first *)
  if R[(I+res) mod MaxReq,(res+J) mod MaxRes] then
  begin
   gnt := (I+res) mod MaxReq;
   S[gnt] := (res+J) mod MaxRes;
   for r := 0 to MaxRes-1 do R[gnt, r] := false;
   for req := 0 to MaxReq-1 do R[req, (res+J) mod MaxRes] := false;
  end;
 for res := 0 to MaxRes-1 do                       (* allocate remaining resources 'least choice first' *)
 begin
  gnt := -1;
  min := MaxRes+1;
  for req := 0 to MaxReq-1 do                      (* find requester with smallest number of requests *)
  begin
   if (R[(req+I+res) mod MaxReq,(res+J) mod MaxRes]) and (nrq[(req+I+res) mod MaxReq] < min) then
   begin
    gnt := (req+I+res) mod MaxReq;
    min := nrq[(req+I+res) mod MaxReq];
   end;
  end;
  if gnt <> -1 then
  begin
   S[gnt] := (res+J) mod MaxRes;
   for r := 0 to MaxRes-1 do R[gnt, r] := false;
   nrq[gnt] := 0;
   for req := 0 to MaxReq-1 do
    if R[req, (res+J) mod MaxRes] then nrq[req] := nrq[req]-1;
  end;
 end;
 I := (I+1) mod MaxReq;
 if I = 0 then J := (J+1) mod MaxRes;
end;
```

APPENDIX C

```
var
  I: 0..MaxRq-1;                            (* round-robin requester offset *)
  J: 0..MaxRs-1;                            (* round-robin resource offset *)
  rq: 0..MaxRq-1;                           (* requester *)
  rs, r: 0..MaxRs-1;                        (* resource *)
  gnt: -1..MaxRq-1;                         (* granted request *)
  min: 0..MaxRs+1;                          (* minimum number of requests *)
  R: array [0..MaxRq-1, 0..MaxRs-1] of boolean;  (* R[i,j] is true if requester i is requesting resource j *)
  NRq: array [0..MaxRq-1] of 0..MaxRs;      (* NRq[i] is the number of resources requested by requester i *)
  Gnt: array [0..MaxRs-1] of -1..MaxRq;     (* Gnt[i] contains grant for requester i *)

procedure arbitrate;

procedure init_resource;
begin
  for rs := 0 to MaxRs-1 do                 (* initialize Gnt *)
    Gnt[rs] := -1;
end; (* init_resource *)

procedure init_requester;
begin
  for rq := 0 to MaxRq-1 do                 (* requesters calculate their priorities *)
  begin
    NRq[rq] := 0;
    for rs := 0 to MaxRs-1 do
      if R[rq, rs] then
        NRq[rq] := NRq[rq] + 1;
  end;
end; (* init_requester *)

procedure resource;
(* input:  rs,                                         *)
(*         R[0..MaxRq-1, (J+rs) mod MaxRs],            *)
(*         NRq[0..MaxRq-1];                            *)
(* output: Gnt[(J+rs) mod MaxRs];                      *)
begin
  gnt := -1;
  if R[(I+rs) mod MaxRq, (J+rs) mod MaxRs] then   (* check round-robin position first *)
    gnt := (I+rs) mod MaxRq
  else
  begin
    min := MaxRs + 1;                       (* find requester with highest priority *)
    for rq := 0 to MaxRq-1 do
      if R[(rq+I+rs) mod MaxRq, (J+rs) mod MaxRs] and (NRq[(rq+I+rs) mod MaxRq] < min) then
      begin
        gnt := (rq+I+rs) mod MaxRq;
        min := NRq[gnt];
      end;
  end;
  if gnt <> -1 then
    Gnt[(rs+J) mod MaxRs] := gnt;
end; (* resource *)
```

APPENDIX C (continued)

```
procedure requester;
(* input:  rq,                                    *)
(*         rs;                                    *)
begin
  if Gnt[(rs+J) mod MaxRs] <> -1 then
    for r := 0 to MaxRs-1 do                (* clear requests of scheduled requester *)
      R[Gnt[(rs+J) mod MaxRs], r] := false;
  NRq[rq] := 0;                             (* requesters calculate their priorities *)
  for r := rs+1 to MaxRs-1 do               (* only consider resources that have not yet been scheduled *)
    if R[rq, (r+J) mod MaxRs] then
      NRq[rq] := NRq[rq] + 1;
end; (* requester *)

begin (* arbitrate *)
  init_resource;
  init_requester;
  for rs := 0 to MaxRs-1 do                 (* schedule resources sequentially *)
  begin
    resource;
    for rq := 0 to MaxRq-1 do
      requester;                            (* requesters update priorities; clear requests when scheduled *)
  end;
  I := (I+1) mod MaxRq;                     (* update offsets for round-robin position *)
  if I = 0 then J := (J+1) mod MaxRs;       (* each requester and resource has its own synchronized copy *)
end; (* arbitrate *)
```

APPENDIX D

```
var
  I: 0..MaxRq-1;                                (* round-robin requester offset *)
  J: 0..MaxRs-1;                                (* round-robin resource offset *)
  rq: 0..MaxRq-1;                               (* requester *)
  rs: 0..MaxRs-1;                               (* resource *)
  gnt: -1..MaxRq-1;                             (* granted request *)
  acc: -1..MaxRs-1;                             (* accepted request *)
  it: 0..MaxIt-1;                               (* iteration *)
  min: 0..MaxRs+1;                              (* minimum number of requests *)
  R: array [0..MaxRq-1, 0..MaxRs-1] of boolean; (* R[i,j] is true if requester i is requesting resource j *)
  NRq: array [0..MaxRq-1] of 0..MaxRs;           (* NRq[i] is the number of resources requested by requester i *)
  NRs: array [0..MaxRs-1] of 0..MaxRq;           (* NRs[i] is the number of requests received by resource i *)
  Gnt: array [0..MaxRs-1] of -1..MaxRq;         (* Gnt[i] contains grant sent by resource i *)
  Acc: array [0..MaxRq-1] of -1..MaxRs;         (* Acc[i] contains grant accepted by requester i *)

procedure arbitrate;

procedure init_resource;
begin
  for rs := 0 to MaxRs-1 do                     (* initialize Gnt *)
    Gnt[rs] := -1;
end; (* init_resource *)

procedure init_requester;
begin
  for rq := 0 to MaxRq-1 do                     (* initialize Acc *)
    Acc[rq] := -1;
  for rq := 0 to MaxRq-1 do                     (* requesters calculate their priorities *)
  begin
    NRq[rq] := 0;
    for rs := 0 to MaxRs-1 do
      if R[rq, rs] then
        NRq[rq] := NRq[rq] + 1;
  end;
end; (* init_requester *)

procedure resource;
(* input:  rs,                                                *)
(*       R[0..MaxRq-1, (J+rs) mod MaxRs],                     *)
(*       NRq[0..MaxRq-1],                                     *)
(*       Acc[0..MaxRq-1];                                     *)
(* output: Gnt[(J+rs) mod MaxRs],                             *)
(*       NRs[(J+rs) mod MaxRs];                               *)
begin
  if (Gnt[(J+rs) mod MaxRs] = -1) or (Acc[Gnt[(J+rs) mod MaxRs]] <> (J+rs) mod MaxRs) then
    if R[(I+rs) mod MaxRq, (J+rs) mod MaxRs] then      (* check round-robin position first *)
      Gnt[(J+rs) mod MaxRs] := (I+rs) mod MaxRq
    else
    begin
      gnt := -1;
      min := MaxRs + 1;                         (* find requester with highest priority *)
      for rq := 0 to MaxRq-1 do
        if R[(rq+I+rs) mod MaxRq, (J+rs) mod MaxRs] and (NRq[(rq+I+rs) mod MaxRq] < min) then
        begin
          gnt := (rq+I+rs) mod MaxRq;
          min := NRq[(rq+I+rs) mod MaxRq];
        end;
      if gnt <> -1 then
```

APPENDIX D (continued)

```
        Gnt[(J+rs) mod MaxRs] := gnt;
      end;
    NRs[(J+rs) mod MaxRs] := 0;
    for r := 0 to MaxRq-1 do                    (* calculate resource's priority *)
      if R[r, (J+rs) mod MaxRs] then
        NRs[(J+rs) mod MaxRs] := NRs[(J+rs) mod MaxRs] + 1;
  end; (* resource *)

  procedure requester;
  (* input:  rq,                                                    *)
  (*       Gnt[0..MaxRs-1],                                         *)
  (*       NRs[0..MaxRs-1];                                         *)
  (* output: Gnt[(J+rs) mod MaxRs],                                 *)
  (*       R[(I+rs) mod MaxRq, 0..MaxRs-1],                         *)
  (*       NRq[(I+rs) mod MaxRq],                                   *)
  (*       Acc[(I+rs) mod MaxRq];                                   *)
  begin
    acc := -1;
    for rs := 0 to MaxRs-1 do                   (* accept round-robin position *)
      if (rq = ((I+rs) mod MaxRq)) and (Gnt[(J+rs) mod MaxRs] = rq) then
        acc := (J+rs) mod MaxRs;
    if acc = -1 then                            (* find resource with highest priority *)
    begin
      min := MaxRq + 1;
      for rs := 0 to MaxRs-1 do
        if Gnt[(J+rs) mod MaxRs] = rq then
          if NRs[(J+rs) mod MaxRs] < min then
          begin
            acc := (J+rs) mod MaxRs;
            min := NRs[acc];
          end;
    end;
    if acc <> -1 then
    begin
      Acc[rq] := acc mod MaxRs;
      for rs := 0 to MaxRs-1 do                 (* clear requests of scheduled requester *)
        R[rq, rs] := false;
    end;
    NRq[rq] := 0;
    for r := 0 to MaxRs-1 do                    (* requesters calculate their priorities *)
      if R[rq, r] then
        NRq[rq] := NRq[rq] + 1;
  end; (* requester *)

  begin (* arbitrate *)
    init_resource;
    init_requester;
    for it := 0 to MaxIt-1 do
    begin
      for rs := 0 to MaxRs-1 do                 (* each resource calculates a grant *)
        resource;                               (* resources operate in parallel *)
      for rq := 0 to MaxRq-1 do                 (* each requester accepts a grant *)
        requester;                              (* requesters operate in parallel *)
    end;
    I := (I+1) mod MaxRq;                       (* update offsets for round-robin position *)
    if I = 0 then J := (J+1) mod MaxRs;         (* each requester and resource has its own synchronized copy *)
  end; (* arbitrate *)
```